SOFTWARE MAINTENANCE: RESEARCH AND PRACTICE
J. Softw. Maint: Res. Pract. **10**, 181–202 (1998)

Practice

An Incremental Approach to Software Systems Re-engineering

MICHAEL R. OLSEM*
USAF Software Technology Support Center, Hill Air Force Base, Ogden UT 84056-5205, U.S.A.

SUMMARY

Software re-engineering can dramatically improve an organization's ability to maintain and upgrade its legacy production systems. But the risks that accompany traditional re-engineering tend to offset the potential benefits.

Incremental software re-engineering is the practice of re-engineering a system's software components on a phased basis, and then re-incorporating those components into production also on a phased basis. Incremental software re-engineering allows for safer re-engineering, increased flexibility and more immediate return on investment. But commercial automation to support incremental software re-engineering is currently weak. In addition, project managers need a methodology to plan and implement software re-engineering projects based on the incremental approach.

This paper covers the advantages of incremental software re-engineering and what is available concerning support technology. The paper describes a process methodology for planning and implementing incremental software re-engineering projects. Finally, gaps in the support technology are identified with suggestions for future tools from vendors. © 1998 John Wiley & Sons, Ltd.

KEY WORDS: incremental re-engineering; evolutionary re-engineering; partial re-engineering; trickle-down re-engineering; revolutionary re-engineering; lump-sum re-engineering

1. INTRODUCTION

1.1. The problem of software maintenance

Most organizations stand at a crossroads of competitive survival—a crossroads created by the information revolution now shaping/shaking the world. Each company or government must either go forward, taking advantage of modern information processing technology, or lag and be bypassed.

Our world runs on the microchip. Our car, TV, pay cheque, medical care, almost

* Correspondence to: Michael R. Olsem, 9401 Lakewood Circle, Norwalk IA 50211–1872, U.S.A.
E-mail: molsem@Q-C-I.com

CCC 1040–550X/98/030181–22$17.50

Received 10 October 1997
Revised 29 December 1997

everything we need and use in our daily life depends on the efficient and reliable processing of data and information. Unfortunately, most of the critical software systems that companies and government agencies depend upon were developed during the same time the computer industry was moving from the toddler stage to adolescence. Tools, methods and technologies have evolved at an incredible pace. We have progressed from room-sized computers proudly displayed behind glass walls, to laptops with many times the computing power of those old behemoths. And this has happened in only 30 to 40 years. Even assuming enlightened organizations have kept their personnel fully trained as each new technology appeared (which is very rare indeed), they are still left with all those critical, difficult to maintain, legacy systems that were developed and still execute on old platforms using chaotic design logic.

There is a synergy between information technology (IT) and an organization's mission critical processes. IT supports business strategies and enables (or constrains) the flexibility to adapt to the dynamic changes of today's political, military, business and economic world. IT can either provide opportunities or impose constraints on process innovations, such as automation, parallel processing, geographical distribution of processing power and the elimination of process intermediaries. Constraints stem from the inherent difficulties of clearly defining the interactions between human and machine. Constraints are also imposed by an IT infrastructure that cannot, or will not, change.

The maintenance of legacy systems is akin to fixing a flat tyre without stopping the vehicle. In many instances, an organization could not survive for the length of time needed to take a mission critical system off line for repairs or upgrade. In general, modifications are implemented, tested within the abilities of the testing infrastructure, and put into production with the fervent hope that the modification was inserted correctly and will not affect all the inter-dependent components of the system. We have compounded these maintenance problems by largely ignoring software maintenance tools and methodologies. When a relatively simple problem (at least from a technical perspective) comes along, such as the two-digit-year millennium problem, the repercussions can be potentially catastrophic with a projected United States Federal Department of Defense (DoD) price tag of $\$7\frac{1}{2}$–15 billion. Even without the millennium problem (now less than two years away), as much as 80% of an organization's software budget now goes toward maintaining legacy systems, thereby severely constraining resources that could otherwise be used to increase competitive advantage.

There are four basic options available to organizations when faced with a legacy system that is consistently difficult to maintain:

- Maintain the status quo: we can usually continue to ignore the problems, but maintenance costs and turn-around times will continue to increase.
- Retire the system: there are many software systems still in production due solely to inertia. Periodically, an organization should assess their legacy systems to determine their usefulness versus the cost to maintain them. The Year 2000 problem has finally forced the DoD to inventory their legacy systems, determine the systems' users and prioritize the systems' importance to each organization's mission. In several instances, it was decided to phase the system out of production.
- Redevelop the system: although an expensive option, at least it has the advantage of

using modern tools, platforms, languages and methodologies—at least for a decade or so after which these new systems may become difficult to maintain in their own right.
- Re-engineer the system: the only technology developed to specifically address the problems of maintaining legacy systems (Davenport, 1993; Rochester and Douglass, 1991; Seymour, 1992). Re-engineering also has the advantage of generally being much cheaper than redevelopment while preserving the mission critical functionality which has allowed these systems to remain in production all these years.

1.2. Software re-engineering

The DoD Joint Logistic Commanders sponsored a workshop in 1992 which, among other met objectives, defined re-engineering terminology for the DoD. The workshop agreed that software re-engineering is the '. . .examination and alteration of a subject system to reconstitute it in a new form and the subsequent implementation of the new form'. A general view of software re-engineering can be found in the volume edited by Robert S. Arnold (1993). In practice, re-engineering (Waters and Chikofsky, 1994) is an umbrella term for the technologies of:

- Reverse engineering: the capture of design information from source code or user interfaces. More than 50% of software maintenance is spent on trying to understand how a legacy system works.
- Forward engineering: processing the resulting design information captured by reverse engineering.
- Redocumentation: the automatic creation of documentation from source code.
- Restructuring: bringing order out of chaotic process logic.
- Retargeting: moving the system to a modern platform or environment.
- Source code translation: translating the system's languages to another language or dialect.
- Data re-engineering: the application of all the above to data files.

1.3. The risks and challenges of system re-engineering

So why do not all software organizations turn to this silver bullet called 're-engineering'? In general it is because the track record of re-engineering projects is not outstanding. Poor planning and over-hyped tools are behind many unsuccessful re-engineering projects. But another key factor is the high risk inherent in traditional re-engineering projects (Johnson, 1997). No organization wants to risk their survival on a single roll of the dice. When an organization replaces a legacy production system with its newly re-engineered counterpart, there is little room for error. Users or customers have always relied on these legacy systems, though perhaps slow and inefficient, to provide a dependable means of manipulating vital information. How often does a new production system, developed from scratch or re-engineered, work the first time, and thereafter, every time? What automated information systems (AIS) manager wants to incur the tremendous pressure created when

a few small errors can make an entire re-engineered system fail? The perceived risk is just too high.

Based on lessons learned from actual re-engineering projects (for example, Byrne (1994)), there are seven major issues facing re-engineering in the near future:

- Integration with business process engineering.
- Targeting non-traditional (non-programmers) users.
- Applying lessons learned to guide future tool selection.
- Integrating heterogeneous tool sets.
- Exploratory 'what if' re-engineering.
- Incremental re-engineering for large systems.
- Cost/benefit analysis to decide whether to re-engineer, redevelop or maintain the current system.

Although this paper will focus primarily on incremental re-engineering of the software of large systems, these other issues will affect, and be affected by, the availability (or lack thereof) of an incremental re-engineering strategy.

2. IMPLEMENTATION STRATEGIES: SYSTEMS VERSUS INCREMENTAL

2.1. Definition of system and incremental re-engineering

The two basic approaches for planning and implementing software re-engineering projects are system re-engineering and incremental re-engineering. The two approaches are characterized by their different methodologies.

- System re-engineering: fully re-engineer the entire system (data files, source code, new platform, etc.), validate the functionality, then implement the newly re-engineered system. For centralized AIS systems, the legacy system is simultaneously retired. For some decentralized embedded systems, the legacy system is phased out during embedded system upgrades. System re-engineering has also been called cold turkey, big bang, revolutionary, or lump-sum re-engineering.
- Incremental re-engineering: also known as evolutionary, chicken little, trickle-down or partial re-engineering, incremental re-engineering uses two methodologies:
  - Re-integrate: re-engineered modules are re-integrated into the legacy system. This is usually applied to small software changes that do not alter the legacy source code language. This tends to limit the applicability of this form of incremental re-engineering to those technologies (such as re-structuring) that do not seriously impact the rest of the legacy system.
  - Not re-integrate: re-engineered modules are not re-integrated into the legacy system. Functionally cohesive modules are identified, isolated, re-engineered and then interfaced with the still non-re-engineered portions of the system. The subsequent system now has both re-engineered and legacy components that interface with each other through a carefully defined mapping mechanism.

2.2. Systems re-engineering

There are several 'official' DoD definitions for system. The Electronic Industries Association (EIA) Standard IS-632 of December 1994, defines a system as 'an integrated composite of people, products and processes that provide a capability to satisfy a stated need or objective'. DoD Regulation 5000.2-R (15 March, 1996) defines a system as 'a combination of elements that shall function together to produce the capabilities required to fulfil a mission need, including hardware, equipment, software or any combination thereof, but excluding construction or other improvements to real property'. These are very broad definitions caused by the wide variety of system types within the DoD (embedded, command and control, and AIS). When practically applied, these definitions allow for wide interpretations. Forced by the two-digit millennium problem, the DoD has (for the first time ever) taken inventory of all their production systems. The Navy reported 3 000 systems, the Army reported 500 systems, and the Air Force reported 2 383 systems. Although each DoD branch would be expected to vary some in the number of production systems, the range from 500 to 3 000 appears to indicate a very different perception of exactly what constitutes a system within the armed services.

For the purposes of this paper, a *system* is a functionally cohesive set of hardware platforms, operating systems, software programs, display terminals, data storage devices, compilers, telecommunications infrastructure and whatever else is necessary for system execution. These components can be classified as either platform, data, application or user interface. Each component may require a different re-engineering strategy and each re-engineering strategy is applicable to specific component classes. Even so, components are interdependent. Real-world re-engineering projects demonstrate that success depends on how well all the interconnected components of the legacy system get migrated. For example, re-engineering application software usually requires re-engineering its data files.

A major disadvantage to the system re-engineering approach is the all-or-nothing high risk of failure. The re-engineered system must contain no errors that affect key user functionality. The system's users expect no less since they have historically relied on the legacy system. In fact, the level of risk for a re-engineered system is perceived as greater than that for a newly developed system. The visibility of a newly developed system is far less than that of a legacy production system. There are fewer expectations by the user community for a new system since no reliance has yet been developed by its users. Users have even come to expect some (minor) shake-out problems for a new system. But this level of tolerance seems to be lacking when the IT staff begins to 'tinker' (from the users' perspective) with existing systems. Errors can be easily introduced during re-engineering due to human intervention, lack of a well-defined process and imprecise tools.

User expectations for a re-engineered system are higher. Although the legacy system may have been quirky, slow or rigid, at least it was working prior to re-engineering. The benefits of re-engineering primarily reside in the future. Better response times, quicker maintenance turn-around, cheaper maintenance, etc., are the long term results of well-planned re-engineering projects. However, future benefits are quickly forgotten in the light of immediate problems. Tolerance for disturbing the users' daily production routine is low. If you replace a working legacy system with its re-engineered counterpart, then the users minimally expect to receive the baseline functionality of the legacy system, regardless

of any additional bells and whistles; most of which are often hidden from the end user anyway.

But there are other reasons making the system re-engineering approach unfeasible in certain instances. Chief among these is the size of the legacy system's components relative to the amount of down time acceptable to the organization. Some insurance companies have investigated re-engineering as a means to migrate their legacy data files to a modern data representation, such as by going to a relational database. But the down time required to move the massive data files from their legacy configuration to the new target configuration was estimated at several weeks. Very few organizations can survive that amount of time without access to their critical data files.

Related to the down time issue, is the problem of parallel maintenance of both the legacy system and its re-engineered counterpart while the re-engineering project is ongoing. Feasible only if the system is small enough, parallel maintenance is a very difficult problem for project management and configuration management. Some functional changes cannot wait until the re-engineered system is in production. Thus, changes to the legacy system must be reflected in the target system. This requires significantly more resources (time, money and personnel). If the legacy system is too large or too dynamic, the organization may not be able to afford the expense of duplicate data files, source code, telecommunications and all the other components that comprise a system.

Finally, owing to the extended amount of time required for large re-engineering projects, the target environment may become obsolete before it is implemented. Five year system re-engineering projects are not unusual. Consider all the technical advances that have occurred over the past five years. Would any user want to be stuck with a 1997 client/server system based around the PC technology of 1992? In addition, the organization's strategic direction will most likely change during the lifetime of a five-year re-engineering project. The system and its functionality may no longer be of strategic importance by the time the re-engineered system can be implemented.

2.3. Incremental re-engineering

Incremental re-engineering has a number of key advantages over system re-engineering:

- Results are more immediate and concrete.
  The users may notice improved response time and/or turn-around time for maintenance. Management is pleased by the short term results and more immediate return on investment (ROI).
- Less risk with better error recovery.
  If an error occurs within the re-engineered component, the entire project is not put at jeopardy. Finding an error within the re-engineered component is easier to pinpoint than an error somewhere in an entire system. For error recovery, simply swap out the new component (replacing it with the legacy component), locate the error and re-install.
- Increased flexibility.
  Organizational strategic direction is always dynamic. External and internal pressures may alter the goals and direction of the overall re-engineering project. Incremental

re-engineering allows an organization to better redirect the re-engineering efforts in response to these changes.

Incremental re-engineering better enables the introduction of functional enhancements, once the component has been re-engineered, validated and implemented. Thus, maintenance need not be postponed while the system is being fully re-engineered and parallel maintenance (of the re-engineered and legacy system) is not required, thus easing the burden on limited resources.

As technology changes, incremental re-engineering can adapt more easily while minimizing the risk of losing the resources already invested in previous re-engineering efforts.

- Project justification is easier.

  Large system re-engineering projects are difficult to justify to management due to the high risk, high cost and uncertain ROI. Incremental re-engineering projects use fewer organizational resources for shorter periods of time. Successful incremental re-engineering projects help to ensure continued management support for future re-engineering.

Many legacy systems were developed incrementally over an extended period of time. So why must we re-engineer these legacy systems as an entire unit in far less time? The reason centres around a lack of tools and methods to support incremental re-engineering. Legacy components need to interface with target components within their own class and across component classes. These issues will be explored later in this paper.

Other considerations regarding incremental re-engineering include increased reliance on configuration management (to track both legacy and target components) and response time degradation due to the extra time required to co-ordinate a response to the user based on data derived from legacy and target components. Response time degradation may be particularly critical for embedded systems.

3. INCREMENTAL RE-ENGINEERING ISSUES

3.1. Gateways

Legacy production systems are composed of four classes of components:

- Software/applications.
- Data files (or sensor input for embedded systems).
- Platforms (hardware, degree of application coupling, operating systems, compilers, etc.).
- Interfaces (user queries and updates originating from external or internal sources).

Components from each class can be re-engineered independently even though all four classes are interdependent. Each class contains unique characteristics, requires different re-engineering strategies, and can represent the entire re-engineering solution (instead of the traditional view of system re-engineering projects that must simultaneously encompass

multiple component classes). Thus, incremental re-engineering allows the flexibility to fix the real problem without having to re-engineer other components involved in ripple effects and interdependencies. For example, if re-engineering the human interface is sufficient, then why not limit the project to that component class?

Gateways are required to interface between components of different classes (interclass) and between components within a class (intraclass). A gateway is a software module introduced between operational software components to mediate between them while masking the internal incremental changes from the users. Since gateways are bidirectional, they are composed of two components, a forward part and a reverse part:

- Forward: a set of logic to convert a request from legacy component A's format/functionality to target component B's format/functionality.
- Reverse: a set of logic to convert a request from target component B's format/functionality to legacy component A's format/functionality.

This conversion may require (depending on the system and gateway) a mapping of data types and structures, signal speeds, handshake protocols, etc. Both the forward and reverse components need logic to determine whether a format/functionality conversion is required since a request between legacy components or between target components will *usually* not require translation (a notable exception occurs during rehosting projects). Otherwise, a message passed from one component to another must map the functionality and format of the sending component to the functionality and format of the receiving component. For example, in Figure 1, if a user submits a database query from a legacy terminal's graphical user interface (GUI), then the gateway must first determine whether the query should be routed to the legacy application component or the target component. If the gateway determines the target component is appropriate, then the gateway must transform the query's legacy format and functionality to the corresponding format and functionality of the target application.

Interclass gateways are more complex than intraclass gateways. Interclass gateways have many more mapping combinations than intraclass gateways. In the preceding illustration, the combinations for the interclass gateway interfacing between the data files and the software applications include:

- legacy applications to legacy data files,
- legacy applications to target data files,
- target applications to legacy data files, and
- target applications to target data files.

In addition, gateway mapping may also need to occur in the reverse direction:

- legacy data files to legacy applications,
- legacy data files to target applications,
- target data files to legacy applications, and
- target data files to target data files.

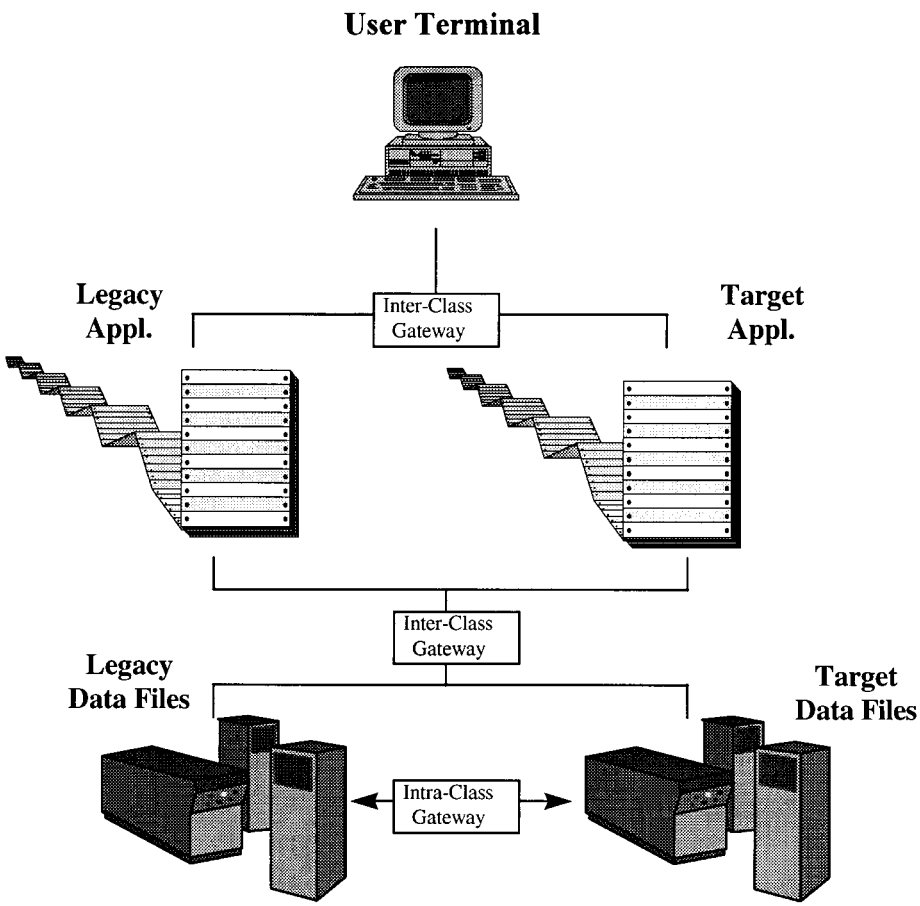

Figure 1. Interclass and intraclass gateways may exist in many different configurations

3.2. Decomposition

A key decision when planning incremental re-engineering projects is how to decompose the legacy system to its components. As part of that decision, the degree of decomposability must be determined. There are different ways to view the decomposition of a legacy system:

- Physical/structural.
  Incremental re-engineering can proceed by hardware components. For example, all the PCs at a given site will be re-engineered, or rehosting will occur before any other re-engineering activity. This approach works best for decentralized systems and many embedded systems.
- Functional/usage/business.
  The usual tactic is to re-engineer components that are mission critical to the

organization and represent a major competitive advantage. Successfully re-engineering critical components early will give the project a high profile with corresponding high benefits. This requires defining the critical functionality then identifying all the components that contribute to it. A strategy to assist in decomposing a system based on functionality is to trace the functionality throughout the source code using impact analysis tools and code slicers (Cimitile, de Lucia and Munro, 1996).

- Degree of inter-connectivity.

  Stove-piped (i.e., stand-alone) systems are easier to decompose and incrementally re-engineer than systems that are tightly coupled with other systems or components used by other systems (such as data files). The drawback to this approach is the tendency to re-engineer non-critical components. If a component is critical, it is generally used by many parts of the organization. For example, an organization's data are much more important than the source code that manipulates the data. But data also tend to be highly interconnected to other systems.
- Data.

  As was mentioned above, an organization's accumulated data, contained within the legacy data files, are far more important than the source code which manipulates that information. Re-engineering the data files before the applications can help identify redundant data (Aiken, Muntz and Richards, 1994). For example, flat files often contain redundant data. These flat files can be re-engineered one at a time (i.e., incrementally) to create a single relational database. A strategy to consider when choosing to decompose the system's data files is to restrict all updates to the target database yet allow query requests to both the target and legacy databases.
- Combination of the above.

  Most project managers appear to base their decomposition decisions on a combination of the above strategies.

In some instances, decomposition is too difficult due to the high degree of coupling between a given component and the rest of the legacy system. If decomposition cannot achieve modularity for the component, there are some alternative strategies (Muller *et al.*, 1993; Sneed and Nyary, 1994). These strategies include encapsulation, repackaging and refining the component's external connections.

4. A PROJECT PROCESS FOR INCREMENTAL RE-ENGINEERING

4.1. Process goals

There are two major stumbling blocks to the practice of incremental re-engineering. One is a lack of support tools. This is covered later in this paper. The other is the lack of a well-defined methodology/process for planning an incremental re-engineering project (Rajlich, Doran and Gudla, 1994). The goals in developing such a process are:

- provide an incremental, flexible approach for upgrading legacy systems,

- improve commonality and modularity across systems,
- improve software re-use and reliability,
- reduce cost and schedule,
- maximize the benefits versus the costs (i.e., optimal ROI),
- facilitate hardware/software evolution,
- reduce long-term maintenance costs, and
- reduce risk.

At some level of detail, re-engineering projects will differ in terms of their processes. Different tasks are needed to achieve different types of changes. A process model created to guide a specific re-engineering project must be based on characteristics of the system to be re-engineered and the changes planned. The collection and sequence of tasks that occur during a project, the structure and semantics of project information, the tools used, and the human roles and relationships that make up the organizational structure of a project, are all project specific. But the basic process of planning and executing an incremental re-engineering project is similar for each project, such that any process variations become significant only later in the project.

A review of published case studies and other literature (for example, Wills, Newcomb and Chikofsky (1995); Wills, Baxter and Chikofsky (1996)) supports our observation that successful incremental software re-engineering projects have 15 steps in common, as described in the next section.

4.2. Steps in an incremental re-engineering process

4.2.1. Develop a strategic plan to define objectives

All fundamental change must come from an organization's strategic objectives. These objectives are expressed in a strategic plan and implemented based on a tactical plan. Thus, incremental re-engineering should be based on an organization's desire for well-defined, incremental change leading to key objectives.

Objectives can be divided into three categories: system, project and organizational. Each category of objectives reflects a different set of concerns about a project and what is to be achieved:

- Organizational objectives specify long and short term goals, legacy system problems addressed by this project, procedures that are to be followed, standards to which project components must adhere, tools that are to be used because the organization supports the use of those tools, etc.
- System objectives typically specify changes to the implementation, design or requirements of a system. System objectives tend to contain action words, such as 'change', 'add', 'replace', etc.
- Project objectives focus on concerns, such as the project duration, staffing details and particular methods to be used. Specific deliverables of the incremental re-engineering project should be stated up front.

4.2.2. Form a core re-engineering team

Within every project there must be an organizational structure to assign project personnel to project roles. Each project member fills one, possibly several, roles within a project. These roles include project manager, team leader, co-ordinator, speciality staff, programmer, database management system (DBMS) administrator, tester, configuration librarian, client, etc. Each role has a title, level of authority and requires a set of qualifications.

The team should include capable representatives from management, users and the software staff. Team members should be technically competent, credible, patient, possess good social/leadership skills and know the organization's culture well enough to function successfully. (Notice that personnel fitting this description are probably some of the organization's best. Technology change, such as re-engineering, requires no less.) The team will also include adjunct members, such as legacy system specialists and support staff.

The core team should initiate a formal and regularly scheduled means of keeping all impacted groups well informed of the incremental re-engineering project's status and how it relates to each interest group, including users, re-engineering team, AIS or embedded staff, and management. Open lines of communication will facilitate co-ordination and co-operation to make the project a success.

4.2.3. Develop a tactical plan

In many re-engineering projects, management develops the tactical plan. But it is our observation that re-engineering projects are more often successful when all the remaining process steps, including developing the tactical plan, are performed by the core re-engineering team. A tactical plan to accomplish the strategic goals identified above would include:

- a description of which components would be replaced,
- the type of re-engineering to be performed on the selected legacy components,
- a listing of the components' relationships with any other components, and
- a schedule for each component's incremental re-engineering, validation and implementation.

After the tactical plan is developed, the remaining steps described below can be addressed and implemented. Some can be done in parallel or done in parallel per re-engineered component or set of components. Any such anticipated parallelism should be noted in the tactical plan. Other steps will need to be repeated and enhanced for each succeeding incremental re-engineering project.

4.2.4. Perform risk analysis, impact analysis and resource analysis

The first task, *risk analysis*, should compare the risks of continuing the status quo (projected over the next several years) against the risks of re-engineering. Risk analysis should rate the importance of the legacy system to the organization's mission and develop a back-up plan in case of project failure. Detecting possible problems before they occur saves time, effort and money later. It may be necessary to re-examine the objectives,

possibly altering or removing objectives that were originally accepted. Modifications to objectives or change requirements should be explained in the 'Risk Report' produced during this process step.

There are a variety of re-engineering project risks. At this point in the process, the main risk is that the amount of effort necessary to implement all identified incremental re-engineering change requirements exceeds the time, resources or staff restrictions under which the project must operate. For example, consider an objective to convert flat data files into a relational DBMS. This objective results in a change requirement to reformat the data files from their existing format into a format supported by the DBMS. This change requirement impacts the portions of the system that read and write to the data files, resulting in additional change requirements to rewrite these sections of code. Further analysis may determine that program logic is based on the organization of the data and, by transforming the data, it will be necessary to modify extensive portions of the program logic. This introduces additional change requirements. Based on the size of the system, what started out as a straightforward change requirement has revealed a family of induced change requirements that will require a great deal of effort to implement. It is not uncommon, once the scope of a change is analysed, to decide either to omit the original objective or to modify it.

The second task, *impact analysis*, should include everything potentially affected (system ripple effects, users and the organization) by the re-engineering project(s), worst case scenarios and a back-up plan for recovery. The impact analysis will most likely discover additional components that need to be taken into account by the incremental re-engineering project. There are commercially available impact analysis tools and source code slicers to assist in this analysis step.

The third task, *resource analysis*, determines whether there is sufficient personnel, funding, time, hardware, infrastructure support, etc. to successfully complete the incremental re-engineering project. A variety of resources may be necessary to support a project. These resources must be identified early. Will key personnel familiar with the legacy system's components be sufficiently available? Key personnel are usually very busy. The system being re-engineered may execute on special equipment. The availability of special purpose hardware can affect system tests and system transition plans. Sometimes a system is re-engineered within one computer environment, such as a workstation, but used in a different environment. Identifying needed resources early in the project simplifies the task of obtaining the resources before they are needed. System resources can be identified by considering several questions. For example, does the system require special purpose hardware to operate? Does equipment need to be allocated to the project, such as computers, terminals or printers? Does the system require special capabilities, such as access to a network?

4.2.5. Employ a configuration management process including support tools

Configuration management (CM) is key to any re-engineering project but is particularly emphasized for incremental re-engineering projects. The organization needs to track legacy and target components and the gateways between components. The objective is to ensure

all production components (legacy and target) are exchanging messages in a form and function expected by the correct version of its interfacing components. This must be accomplished in spite of the parallel re-engineering of different system components.

4.2.6. Design and install the target environment

The U.S. Air Force's Software Technology Support Center (STSC) views software re-engineering as the means to migrate legacy systems from an outdated/inefficient maintenance environment to the newly defined target environment based on the Software Engineering Institute's Capability Maturity Model (CMM) (Olsem, 1995). The CMM contains a list of key process areas (KPA) associated with five software development maturity levels. Each KPA is satisfied by the organization in a way unique to the organization's needs and strategic goals. An organization should describe in its strategic plan how it wants to develop/maintain systems in accordance with the CMM. Thus, the target environment should incorporate the implementation strategy for each KPA, but the tactical plan goes beyond the CMM to describe the target hardware environment and development paradigm (object-orientated, structured, client/server, etc.) necessary to support the transition of production software to this CMM-based development/maintenance environment (Mittra, 1995). Thus, it is important to define the target system's platform, languages, interfaces, development paradigm, data files, tools, etc., even though the immediate incremental re-engineering project may not encompass all of the system's components.

4.2.7. Define, collect and analyse metrics

'And everything else . . . whether it be technology, whether it be new tools such as I-CASE or whether it be process improvement, whatever it might be in the final analysis, they mean nothing if we can't measure and document the level of improvement, and in fact you will not be able to sell to management those benefits unless you can demonstrate those savings….' according to Lloyd K. Mosemann (1993).

To get to some place, a person must know both what is the starting point as well as what is the destination. Metrics can give a baseline to measure the results of the incremental re-engineering project(s). Metrics should directly correspond to the strategic goals and objectives defined earlier (see the strategic planning step for types of goals/objectives). These metrics should also include a means to measure the project's status/progress during each incremental project phase.

4.2.8. Identify, inventory, analyse and decompose legacy system components

This step is critical for the success of incremental re-engineering projects, and must include both soft and hard artefacts. The decomposition strategy chosen for the components directly affects all other aspects of the re-engineering project(s). For more information, refer to the Decomposition section (3.2) earlier in this paper.

For each system artefact, it is necessary to identify and collect the components that

comprise that system support the system or describe it. There are a variety of components to be collected, including: programs, online screens, job control scripts, data stores, databases and library routines. There are also supporting components, such as user manuals, reference guides, system requirement documents, design documents, maintenance history records, etc. Other supporting components include test plans which consist of test cases, test input data and logged test output data.

Relationships between components must be determined during this process step. These relationships are stored as attributes of the related components. One such relationship is the 'depends on' attribute. This attribute indicates that one component depends on another component. For example, a document that describes module interfaces in a program depends on that program. The document also depends on the specific modules documented. If the modules are changed, it may be necessary to change the documentation as well.

4.2.9. Develop/enhance testing strategies and testbed

Each incrementally re-engineered component should be completed and validated before moving on to the next. The goal of testing, as applied to re-engineering projects, is to maintain the baseline behaviour/functionality of the legacy system before, during and after the implementation of incremental re-engineering changes. The testing objective for such regression testing is deviation detection and deviation identification.

A set of tests to validate the legacy system's baseline functionality must be fully developed prior to re-engineering, so that the same tests can be run as regression tests to validate the functionality of the re-engineered system component as well as the functionality of the legacy components as they interact with the re-engineered components. This is a key reason why successful re-engineering projects try to minimize functional enhancements until after validation of the re-engineered component or system. Adding functionality tends to invalidate this set of baseline tests. Therefore, if no new functionality is introduced, the routines and variables before and after re-engineering can be mapped to their counterparts as a prelude to white box testing. This is true whether the re-engineering involves retargeting, restructuring, translation, reverse engineering forward engineering or redocumentation. The process flow (i.e., sequence of routines) may be radically altered during re-engineering (due perhaps to restructuring) but the underlying logical implementation of system requirements remains constant.

Black box testing is normally insufficient for re-engineering projects because, at most, one can know an error occurred but not know what caused it. Because of re-engineering changes to the system's control flow, language, structure, etc., it becomes very difficult to associate an error with a given part of the system using black box testing methods.

To maintain core functionality, systems before and after re-engineering have a set of corresponding routines and variables. By linking these routines and variables in the before and after versions, the testing personnel can trace potential errors by monitoring variable values in a white box testing manner. This can filter out any extraneous information regarding other variables or routines by giving priorities and classes to existing variables and routines. This white box testing method contributes to traceability.

4.2.10. Identify appropriate and available tools and supporting technology

Tools must be chosen based on the project's strategic goals/objectives and available resources (funding, personnel, and time). Too often, tools are chosen first then plans are developed around the functionality and constraints of the tools. This is the tail wagging the dog. To avoid that difficulty requires resisting the vendors' sales pitches and shopping for a suite of re-engineering tools that will accomplish the organizational, system and project goals outlined by the first step of this process.

4.2.11. Train personnel

Traditionally, personnel need training in tools, methods, gateways, etc. But personnel should also be trained in managing technology change and the goals/objectives of the incremental re-engineering project. A typical approach is to train the core re-engineering team and then use them to train additional personnel. Ignorance creates opposition and possibly sabotage (either inadvertently or though non-co-operation).

4.2.12. Plan, design and implement a pilot project

The pilot project will study methods, tools and results. When designing the pilot project, the personnel should try to avoid the pitfalls common to pilot projects in general. These include ignoring scaling issues (ensuring the method and tools will also work on large production systems) and the microscope or Hawthorne effect (high profile, highly studied projects tend to succeed due to their visibility).

Lessons learned by the pilot project must be carefully recorded, studied and extrapolated to larger scale incremental re-engineering projects. This is particularly true when the pilot project is evaluating re-engineering tools. Thus, all the metrics defined to support the strategic and tactical plans should be applied to this pilot project.

4.2.13. Enhance and create documentation for different layers of abstraction

With documentation, the personnel must always walk a fine line between documentation critical for maintenance, without creating excessive documentation that will be burdensome, ignored or rapidly obsoleted. The decision is usually unique to the organization's requirements. But, a 'correct' means or way (in the sense of the organization's requirements) to document the re-engineered components is critical to achieving the long term benefits the organization hopes to reap.

For incremental re-engineering, documentation requires additional flexibility due to the dynamic nature of succeeding incremental projects, parallel development efforts on different components, and the need to smoothly integrate the combined documentation of lower level components to represent higher levels of abstraction.

In recent experiments, layered documentation proved much more flexible and effective (as compared with embedded comments or non-layered annotations) in communicating

system design, purpose, domain, etc. Three layers are key: domain, algorithm and representation. Each layer corresponds to a level of abstraction:

- Domain: an overview including block diagrams, data flow diagrams (or object model), program architecture and interfaces (GUI, reports, input transactions, etc.),
- Algorithms: plain text to mirror the formal algorithms, and
- Representation: describes the variables and their relationships, such as a database model.

Other dimensions that should be tracked include constraints, operations, author, time, error history, tests and warnings.

A strategy that is gaining credence is the maintenance of systems at the design level. Following reverse engineering, some re-engineering tools allow maintenance personnel to manipulate the graphical design representation followed by forward engineering to create executable source code. This approach ensures the design information is always current. Other recent documentation tools employ hypertext and web technology.

4.2.14. Re-engineer the selected incremental components

The preceding process steps assist an organization to select and analyse appropriate components for incremental re-engineering. This step simply performs the indicated re-engineering, whether it be code translation, re-structuring, reverse engineering, re-targeting or data re-engineering.

4.2.15. Migrate re-engineered components into production

The gateways, re-engineered data files and re-engineered software are all placed into production in accordance with the CM plan.

5. TECHNOLOGY SUPPORT

5.1. Existing tools

The USAF Software Technology Support Center (STSC) maintains a database of software re-engineering, testing and code analysis tools. The database of re-engineering tools can be downloaded from the STSC web site at: *http://www.stsc.hill.af.mil*

This web site is updated once every three months from the more dynamic, in-house database of tools and services. Free to anybody contacting the STSC, a customized list of tools and services can be created, based on an organization's platform, languages, price range, type of re-engineering, etc. Information can be obtained by contacting the STSC's database administrator (currently Karen Rasmussen) via telephone (801-825-2655) or via email at: *rasmussk@software.hill.af.mil*

The STSC maintains a database of analysis tools and testing tools in addition to tracking tools in the following re-engineering categories:

Table 1. Database gateways

Product	Vendor	Phone number
Database Gateway	Micro-Decisionware	303-443-2706
EDA/SQL	Information Builders, Inc.	212-736-4433
Enterprise Connect	Sybase	800-879-2273
Gupta/SQLHost	Gupta	650-321-9500
Informix Gateway	Informix Software, Inc.	650-926-6300
Ingres/Gateway	Ingres	510-769-1400
COBOL/SQL Transparency System	MicroFocus	800-872-6265
SQL*Connect & SQL*Net	Oracle	650-506-7000
Platinum Integrator	Platinum Technology	800-442-6861
Transparency	Computer Associates	516-342-5224
XDB	XDB Systems, Inc.	301-317-6800

Table 2. Application gateways

Product	Vendor	Phone number
Copernicus	New Paradigm Software	212-557-0933
TransAccess	Netwise	303-442-8280

- reverse engineering,
- forward engineering,
- re-documentation,
- re-structuring,
- re-targeting, and
- data re-engineering

What the STSC does *not* yet track is the availability of commercial gateways. The condensed lists of existing gateways and interface products shown in Tables 1, 2 and 3 are adapted from Brodie and Stonebraker (1995).

Table 3. Interface (graphical display) gateways

Product	Vendor	Phone number
Enfin	Easel	617-221-2100
FlashPoint	Knowledge Ware	800-444-8575
Mozart Composer	Mozart Systems	650-340-1588

5.2. Development of new tools

5.2.1. Tool ideas for incremental re-engineering

There are a number of serious gaps in incremental re-engineering support technology. In addition, there are some serious technology gaps for software re-engineering in general. We shall begin with a list of tools that should be developed to enable incremental re-engineering:

- Gateways.
  As can be seen from Tables 1, 2 and 3, there are very few commercial gateways. The largest group of these are database gateways most of which allow only for the migration of CICS/IMS to SQL/DB2. Incremental re-engineering needs specific gateways between system components and generic gateways between classes of components. Interclass gateways allow the four classes of system components (platform, data, user interfaces and applications) to exchange information between legacy and re-engineered components. Intraclass gateways allow components of the same class to exchange information.
- Intelligent project management tools.
  Incremental re-engineering allows for greater flexibility in the planning and scheduling of the overall re-engineering project. Similarly, better project management tools will be required to plan, manage, schedule and estimate the cost of the incremental re-engineering projects, both for stand-alone use and for integrated use into higher levels of abstraction. Such project management tools should allow for incremental re-engineering projects developed in parallel and designed for parallel processing (using multithread and multitasking techniques).
  For example, we need project management tools to manage migration of centralized mainframe systems to distributed client/server systems. This conversion requires both parallel transition efforts (via incremental re-engineering) and parallel processing techniques for co-ordination of file updates, etc.
- Analysers.
  Although there are currently many source code analysers on the market, incremental re-engineering requires some specialized analysis tools. It would be useful to have a tool to determine a component's decomposability based on identified architectural components and their dependencies/interconnections. Such a tool would identify and assess potential migration difficulties leading to a metric that measures the degree of a component's portability.
  Another useful analysis tool would be a traceability mapping tool. Such a tool would link component functionality to specific code modules before and after re-engineering.
  Finally, in support of an organization's migration from mainframe based systems to client/server systems (a popular re-engineering goal), an analysis tool would be handy in identifying common legacy components and assisting in the optimal distribution of code and data.
- Encapsulation tools for non-decomposable components.

Some components cannot be decomposed because they are simply too complex or too interconnected (i.e., tightly coupled). Instead, these components can be encapsulated or wrapped. Encapsulation is a re-engineering technique whereby the component is left unchanged but all input and output is carefully controlled by a front-end filter. Primarily used on entire programs and systems, encapsulation can be scaled down and applied to incremental parts of the legacy data, application or user interface. If the appropriate analysers existed, the decision to encapsulate could be based on a given threshold of portability.

5.2.2. Tool ideas for software re-engineering

We now briefly discuss tools that should be developed for any software re-engineering project, whether systems or incremental.

- User interface re-engineering tools.

  Although some research is being done at Georgia Institute of Technology (Moore, 1996), the author is unaware of any commercially available user interface re-engineering tools.
- Better database re-engineering tools.

Tools are needed that can automatically generate a database schema (relational or other) based on an analysis of all legacy source code (legacy and re-engineered) that interfaces with the legacy data files. Such a tool may also be able to create a high level data model for the organization.

Just as there are tools that can detect dead or unstructured source code, similarly we need tools to detect data definition inconsistencies. We also need tools to assist in the integration of databases based on the resulting schemas defined by the preceding data modelling tools (perhaps called ‘schema mappers’).

- Web-based tools.

  One of the defining events of the information age was the development and popularization of the world-wide web. Organizations are beginning to use the web and intranets to develop distributed and client/server computing systems. With the exception of a few tools, such as Hyperbook 2.0 (from Computer Command & Control Company, Philadelphia PA), the software re-engineering vendor community has been slow in taking advantage of web technology.
- Implement re-engineered/modified components while the system is in production.

  Some research is being performed that would allow organizations to switch from a legacy component to its next version without taking the component off line. The re-engineered/modified component would work in parallel with the legacy component, gradually taking over the workload until such time as the legacy component was no longer necessary. Then the legacy component could be safely removed or left in place as a back-up component should a problem arise in the new component. A switching mechanism would allow the software staff to direct the production control flow with an indicator that would show when the legacy component was no longer needed.
- Tool integration.

Large and complex software re-engineering projects always involve multiple types of re-engineering with corresponding tools. To ensure that these tools can exchange standard data, re-engineering projects must rely on larger vendors that offer several different re-engineering tools even though, for some aspects of the project, a competitor's tool may be better suited. Tool integration has been attempted by the DoD's I-CASE project with mixed results. Instead of forcing vendors to fit a base architecture, it may be easier to use a common meta-language or standard data exchange (similar to CDIF for CASE tools). Whatever method is used, software re-engineering needs a means to integrate multiple tools easily from different vendors.

6. CONCLUSION

Traditional full system software re-engineering is being rejected by more and more organizations due to high risks, large up-front costs, and an inability to include mid-project changes in requirements or technology. In many instances, full software system re-engineering is not even an option for an organization's larger, mission critical software systems.

Incremental software re-engineering controls the level of risk. Management and the user community can obtain a more immediate return on investment and incremental re-engineering allows for changes to be smoothly incorporated after the completion of any stage of the project.

Incremental re-engineering requires a means to interface between re-engineered components and non-re-engineered components. Such gateways are not widely available from commercial vendors. This should be an opportunity for the vendor community if incremental re-engineering was more widely embraced by project managers, but project managers will not attempt an incremental approach to re-engineering without a process template to guide their efforts.

The process described in this paper is the result of an analysis of many DoD and non-DoD re-engineering projects—what worked, what did not and why. This process should give project managers the confidence to at least explore the incremental approach to re-engineering. When enough project managers realize that incremental re-engineering is a preferable alternative, vendors will quickly recognize a new potential market for support technology.

References

Aiken, P. H., Muntz, A. and Richards, R. (1994) 'DoD legacy systems: reverse engineering data requirements', *Communications of the ACM*, **37**(5), 26–41.

Arnold, R. S. (Ed) (1993) *Software Re-engineering*, IEEE Computer Society Press, Los Alamitos CA, 688 pp.

Brodie, M. L. and Stonebraker, M. (1995) *Migrating Legacy Systems: Gateways, Interfaces & the Incremental Approach*, Morgan Kaufmann Publishers, Inc., San Francisco CA, 210 pp.

Byrne, E. (1994) 'Generating project–specific re-engineering process models', in *Proceedings of the 6th Annual DoD Software Technology Conference*, USAF Software Technology Support Center, Hill Air Force Base, Ogden UT, available on CD–ROM.

Cimitile, A., de Lucia, A. and Munro, M. (1996) 'A specification driven slicing process for identifying reusable functions', *Journal of Software Maintenance*, **8**(3), 145–178.

Davenport, T. H. (1993) *Process Innovation: Re-engineering Work Through Information Technology*, Ernest Klett Verlag, Stuttgart, 337 pp.
Johnson, R. E. Jr. (Ed) (1997) *Software Re-engineering Assessment Handbook*, Version 3.0, JLC/JGSE R&R FWG, Washington DC, available from USAF Software Technology Support Center, Hill Air Force Base, Ogden UT, via http://www.stsc.hill.af.mil
Mittra, S. S. (1995) 'A road map for migrating legacy systems to client/server', *Journal of Software Maintenance*, **7**(2), 117–130.
Moore, M. (1996) 'Rule–based detection for reverse engineering user interfaces', in *Third Working Conference on Reverse Engineering*, IEEE Computer Society Press, Los Alamitos CA, pp. 42–48.
Mosemann, L. K. (1993) 'Software technology conference closing address', *CrossTALK*, **6**(13 special issue), 2–5.
Muller, H. A., Orgun, M. A., Tilley, S. R. and Uhl, J. S. (1993) 'A reverse–engineering approach to subsystem structure identification', *Journal of Software Maintenance*, **5**(4), 181–204.
Olsem, M. R. (1995) '*STSC re-engineering technology report*', Document No. TRF–RE–9510–00.04, USAF Software Technology Support Center, Hill Air Force Base, Ogden UT, available via http://www.stsc.hill.af.mil.
Rajlich, V., Doran, J. and Gudla, R. T. S. (1994) 'Layered explanations of software: a methodology for program comprehension', in *Proceedings of the 3rd Workshop on Program Comprehension*, IEEE Computer Society Press, Los Alamitos CA, pp. 46–52.
Rochester, J. B. and Douglass, D. P. (Eds) (1991) 'Re-engineering existing systems', *I/S Analyzer*, **29**(10), 1–12.
Seymour, P. L. (1992) 'Tutorial 3: software re-engineering in the real world', in *Conference Proceedings 9th International Conference and Exposition on Software Maintenance & Re–engineering*, United States Professional Development Institute, Silver Spring MD, 15 pp.
Sneed, H. M. and Nyary, E. (1994) 'Downsizing large application programs', *Journal of Software Maintenance*, **6**(5), 235–247.
Waters, R. G. and Chikofsky, E. (1994) 'Reverse engineering: progress along many dimensions', *Communications of the ACM*, **37**(5), 22–25.
Wills, L., Newcomb, P. and Chikofsky, E. (Eds) (1995) *Proceedings 2nd Working Conference on Reverse Engineering*, IEEE Computer Society Press, Los Alamitos CA, 352 pp.
Wills, L., Baxter, I. and Chikofsky, E. (Eds) (1996) *Proceedings 3rd Working Conference on Reverse Engineering*, IEEE Computer Society Press, Los Alamitos CA, 312 pp.

Author's biography:

Mike Olsem is a Senior Systems Engineer at SAIC assigned to work at the U. S. Air Force Software Technology Support Center (STSC). He specializes in applying software engineering technology, in automating software maintenance, in software tools, and in the year 2000 problem. His B.S. degree is in physics from Iowa State University, and his M.S. degree is in computer science from Brigham Young University in Utah. His e-mail address is: molsem@Q-C-I.com.

